UNIVERSITAT POLITÈCNICA DE CATALUNYA

Tema 5: Diseño del Juego de Instrucciones

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya





Índice

- Tipos de Máquinas
- Modos de direccionamiento
- Estudio Cuantitativo del nivel LM
- RISC vs CISC
- μprogramación
- Procesadores x86

Introducción

Una instrucción de Lenguaje Máquina es una tira de bits que especifica:

- **Código de operación:**
  - Operación a realizar
- **Modos de direccionamiento:**
  - Dónde localizar los operandos
  - Dónde dejar el resultado
- **Secuenciamiento:**
  - Cuál es la siguiente instrucción a ejecutar
  - Generalmente es implícito
  - Instrucciones de Salto, llamada/retorno subrutinas

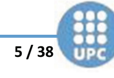
Introducción

- **Criterios de diseño de las instrucciones:**
  - Instrucciones cortas mejor
  - Longitud suficiente
  - Longitud múltiplo de la unidad mínima de direccionamiento (byte)
- **Formato de las instrucciones:**
  - Fijo
    - ✓ Decodificación rápida y fácil
    - ✓ Desperdicio de memoria
  - Variable
    - ✓ Aprovechamiento de memoria
    - ✓ Decodificación compleja

Introducción

- **Tipos de Arquitecturas** (en función de los **operandos explícitos**)

#operandos (memoria)	Tipo de Arquitectura		Ejemplo: A = A + B	Procesador
0 (0)	Pila		PUSH A PUSH B ADD POP A	HP3000, T800
1 (1)	Acumulador		LOAD B ADD A STORE A	i8080
2 (1)	General Purpose Register (GPR)	Registro /Memoria	MOV B,R1 ADD R1,A	x86
3 (0)		Load / Store	LOAD A,R1 LOAD B,R2 ADD R1,R2,R3 STORE R3,A	Alpha 21264
3 (3)		Memoria / Memoria	ADD A,B,A	VAX-11



Máquina de PILA

- Sin operandos explícitos a excepción de las instrucciones de salto y acceso a memoria.

Aritméticas, lógicas y comparación	ADD, SUB, MUL, DIV, AND, OR, XOR, CMPge, CMPeq, ...	pila[TOP] ← pila[TOP] op pila[TOP+1] TOP ← TOP+1
Acceso Memoria	PUSH @	TOP ← TOP-1 pila[TOP] ← M[@]
	POP @	M[@] ← pila[TOP] TOP ← TOP+1
Salto	Bcond $	if (pila[TOP]) PC ← PC + despl TOP ← TOP+1
	BR $	PC ← PC + despl

- Procesadores antiguos en los que la memoria era un recurso escaso. Máquina Virtual Java.
- La pila es un recurso hardware (equivalente al banco registros coma flotante x86)
- ↑ Instrucciones muy cortas, buena densidad de código.
- ↓ La pila no se puede acceder aleatoriamente, es difícil generar código eficiente.
- ↓ La pila puede ser un cuello de botella.

Máquina de ACUMULADOR

- Dispone de un acumulador como operando implícito de todas las operaciones

Aritméticas, lógicas y comparación	ADD @ AND @ CMPge @	ACC ← ACC op M[@]
Acceso Memoria	LOAD @	ACC ← M[@]
	STORE @	M[@] ← ACC
Salto	Bcond $	if (ACC) PC ← PC + despl
	BR $	PC ← PC + despl

- Acumulador es un termino arcaico para referirse a registro.
- ↑ Estado interno de la máquina mínimo (1 registro)
- ↑ Instrucciones cortas
- ↓ Tráfico con memoria muy elevado

Arquitecturas GPR

Arquitecturas con Registros de Propósito General (GPR, General Purpose Register)

- Características que dividen a las arquitecturas GPR:
  - 2 o 3 operandos:
    - ✓ OP2 ← OP1 (op) OP2
    - ✓ OP3 ← OP1 (op) OP2
  - ¿Cuántos de estos operandos pueden estar en memoria?
    - ✓ Desde ninguno en las máquinas Load/Store
    - ✓ Hasta todos en las máquinas Memoria/Memoria
- ↑ Modelo más general para la generación de código, código eficiente
- ↑ Acceso rápido a los registros
- ↑ Utilización eficiente de los registros por el compilador
- ↑ Reducción del tráfico con memoria
- ↓ Todos los operandos son explícitos, algunas instrucciones pueden ser muy grandes.

- Las arquitecturas GPR son las más adecuadas y extendidas. Interesa que los registros sean equivalentes (**ortogonalidad**) y numerosos.

Máquina de Registro / Memoria

- Procesador con dos operandos explícitos, uno de ellos puede estar en memoria.

Aritméticas, lógicas y comparación	**ADD op1, op2** **AND op1, op2** **CMP op1, op2**	**op2 ← op2 op op1**
Salto	**Bcond $**	**if (cond) PC ← PC + despl**
	BR $	**PC ← PC + despl**

- Uno de los operandos es simultáneamente fuente y destino.
- Aparecen instrucciones de movimiento de datos.
- ↑ Los datos son accesibles sin lectura previa de memoria
- ↓ Los operandos no son equivalentes (un operando fuente se modifica)
- ↓ Codificar una dirección y un registro puede limitar el número de registros
- ↓ Los ciclos por instrucción varían dependiendo del tipo de acceso (de 0 a 2 accesos a memoria en 1 única instrucción).

Máquina de Memoria/ Memoria

- Procesador con 3 operandos explícitos, cualquiera de ellos puede estar en memoria.

Aritméticas, lógicas y comparación	**ADD op1,op2,op3** **AND op1,op2,op3** **CMP op1,op2,op3**	**op3 ← op2 op op1**
Salto	**Bcond $**	**if (cond) PC ← PC + despl**
	BR $	**PC ← PC + despl**

- ↑ Código muy compacto
- ↑ No hace falta utilizar registros para variables temporales
- ↓ Diferentes tamaños de instrucción ⇒ Dificulta la búsqueda y decodificación de instrucciones
- ↓ Diferentes cargas de trabajo por instrucción
- ↓ La memoria se convierte en el cuello de botella

Máquina Load/Store

- Procesador con 3 operandos explícitos en registros.

Aritméticas, lógicas y comparación	ADD Ri, Rj, Rk AND Ri, Rj, Rk CMP Ri, Rj, Rk	Rk ← Ri op Rj
Memoria	LOAD D(Rj),Rk	Rk ← M[Rj+D]
	STORE Rk,D(Rj)	M[Rj+D] ← Rk
Salto	Bcond Ri, $	if (cond(Ri)) PC ← PC + despl

- Todas las operaciones aritméticas se realizan sobre registros.
- Necesita instrucciones específicas para acceder a memoria (load / store)
- ↑ Instrucciones codificadas de forma fija ⇒ Facilita la búsqueda y decodificación de instrucciones
- ↑ Generación de código sencilla (el compilador tiene pocas alternativas)
- ↑ Todas las instrucciones tardan tiempos parecidos.
- ↓ Hacen falta más instrucciones, p.e. las utilizadas para acceder a memoria
- ↓ Formato fijo ⇒ Desperdicio de memoria

Ejemplo: evaluar una expresión aritmética

ARQUITECTURA	PILA	ACUMULADOR	REGISTRO / MEMORIA	MEMORIA / MEMORIA	LOAD / STORE
D = (A+B*C)/A	push B push C mul push A add push A div pop D	load B mul C add A div A store D	mov B,R0 mul C,R0 add A,R0 div A,R0 mov R0,D	mul B,C,R0 add A,R0,R0 div A,R0,D	load B,R0 load C,R1 mul R0,R1,R2 load A,R0 add R0,R2,R3 div R0,R3,R4 store R4,D
Instrucciones	8	5	5	3	7
Accesos a Memoria	5	5	5	5	4

Ejemplo: evaluar una expresión aritmética

ARQUITECTURA	PILA	ACUMULADOR	REGISTRO / MEMORIA	MEMORIA / MEMORIA	LOAD / STORE
`A = (A-B*C)/(D+E)`	push E push D add push C push B mul push A sub div pop A	load B mul C store tmp load A sub tmp store A load D add E store tmp load A div tmp store A	mov D,R1 add E,R1 mov B,R2 mul C,R2 sub R2,A div R1,A	add D,E,R1 mul B,C,R2 sub A,R2,R2 div R2,R1,A	load D,R1 load E,R2 load B,R3 load C,R4 load A,R5 add R1,R2,R6 mul R3,R4,R7 sub R5,R7,R8 div R8,R6,R9 store R9,A
Instrucciones	10	12	6	4	10
Accesos a Memoria	6	12	6	6	6

Ejemplo: un código simple en C

ARQUITECTURA	PILA	ACUMULADOR	REGISTRO / MEMORIA	MEMORIA / MEMORIA	LOAD / STORE

while(A!=B){
if (A>B)
A=A-B;
else
B=B-A;
}
	W:push A
	push B
	cmpne
	Bfalse end
	push B
	push A
	cmpg
	Bfalse E
	push B
	push A
	sub
	pop A
	br W
	E:push A
	push B
	sub
	pop B
	br W
	end:
		W:load A
		cmpne B
		Bfalse end
		load A
		cmpg B
		Bfalse E
		load A
		sub B
		store A
		br W
		E:load B
		sub A
		store B
		br W
		end:
			W:mov A,R1
			cmp B,R1
			Je end
			Jle e
			SUB B,R1
			mov R1,A
			br W
			E:sub R1,B
			br W
			End:
				W:cmp B,A
				je end
				jle e
				sub A,B,A
				br W
				E:sub B,A,B
				br W
				end:
					load A,R1
					load B,R2
					W:Seq R1,R2,R3
					JNE R3,end
					Sle R1,R2,R3
					JNE e
					sub R1,R2,R3
					br W
					E:sub R2,R1,R3
					br W
					end:
					store R1,A
					store R2,B

Inst. Estáticas	18	14	9	7	12
Inst. Dinámicas	13·Niter + 4	10·Niter + 3	(6 o 7)·Niter + 3	5·Niter + 2	6·Niter + 6
Accesos a Memoria	7·Niter	8·Niter	(3 o 4)·Niter	5·Niter	4

Modos de Direccionamiento

- **Algoritmo** utilizado para acceder a los **operandos** de una instrucción. Para codificar el modo necesitamos 2 campos (implícitos o explícitos):
  - Modo de direccionamiento: indica el algoritmo a utilizar para calcular la dirección efectiva
  - Una serie de valores, que pueden ser:
    - ✓ Dirección
    - ✓ Registro
    - ✓ Desplazamiento
    - ✓ Inmediato
- Criterios de diseño
  - Acceso a todo el espacio de direcciones
  - Acceso eficiente a estructuras de datos
  - Facilitar la comunicación con subrutinas

Modos de Direccionamiento

- **Modo registro.**
  - El operando se encuentra en uno de los registros del procesador
  - ↑ Acceso rápido
  - ↑ Pocos bits
  - ↓ Se necesitan instrucciones para mover datos con memoria
  - Ejemplo: movl %edx, %eax
- **Modo inmediato.**
  - El operando se encuentra en la instrucción
  - Indispensable para trabajar con constantes
  - Ejemplo: movl $34, %eax
- **Modo Absoluto** (Directo).
  - La dirección del operando se encuentra en la instrucción
  - ↓ Se necesitan muchos bits para codificar la dirección
  - Ejemplo: movl 1242451, %eax

Modo registro indirecto.

- La instrucción codifica el registro que contiene la dirección del operando
- ↑ Pocos bits
- Ejemplo: movl (%ebx), %eax
- ↑ Posibilidad de autoincremento / autodecremento para facilitar accesos a arrays, pila, etc. (p.e. VAX11):
  - ✓ Autoincremento: MOVL R1, (R2)+
  - ✓ Autodecremento: MOVL R1, -(R2)

R

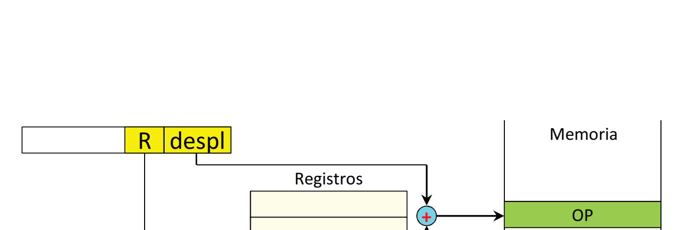
Registros

Memoria

OP

Modo Base + desplazamiento

- La dirección se calcula utilizando un registro y un desplazamiento (±)
- Si el desplazamiento es cero, equivale al modo registro indirecto
- Ejemplo: movl -24(%ebp), %eax
- ↓ Cálculo de la dirección





Modo indexado.

- La dirección efectiva se obtiene sumando el contenido de dos registros
- ↑ Pocos bits
- ↑ Útil en el acceso a vectores
- ↓ Cálculo de la dirección
- Ejemplo: movl (%ebx, %esi), %eax

Ri Rj

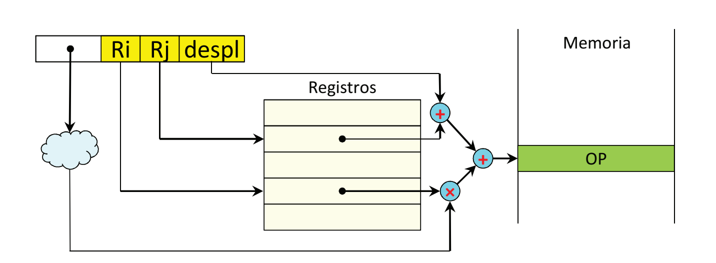
Registros

Memoria

OP

Modo escalado

- Especialmente útil para acceder a vectores.
- El escalado depende de la instrucción
- ↓ Cálculo de la dirección
- Ejemplo: movl -44(%ebx, %esi, 4), %eax



Modos de Direccionamiento

- **Se pueden implementar «modos combinados»**
- **Post - indirección**
  - El dato obtenido con los modos anteriores es la dirección del operando.
  - Ejemplo: **Base + desplazamiento post-indirecto**
  - Ejemplo: MOVL @32(SP),R3

R | despl

Registros

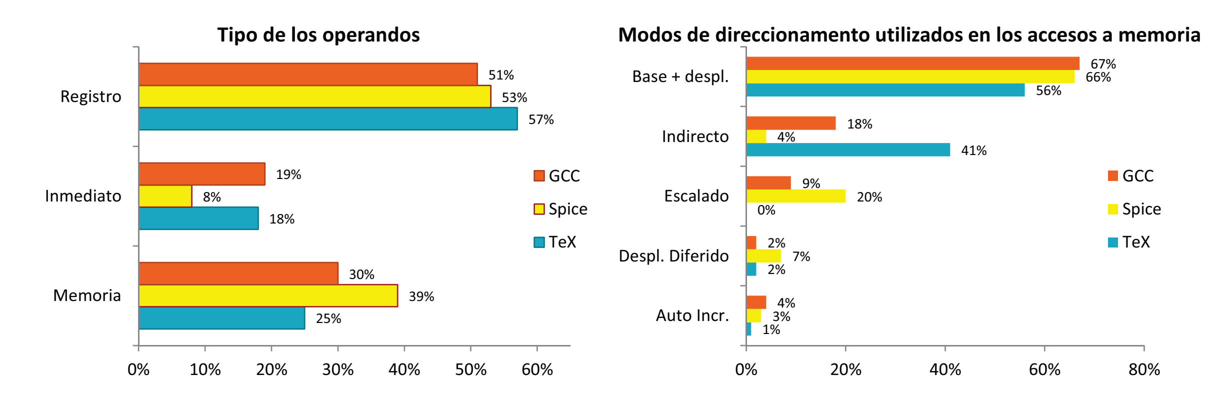
Memoria

OP

- **Post - escalado**
  - El VAX11, permitía combinar el modo escalado (indexado en terminología de DIGITAL) a cualquier otro.
  - Ejemplo: MOVL @32(SP)[R7],R3

Estudio Cuantitativo Lenguaje Máquina

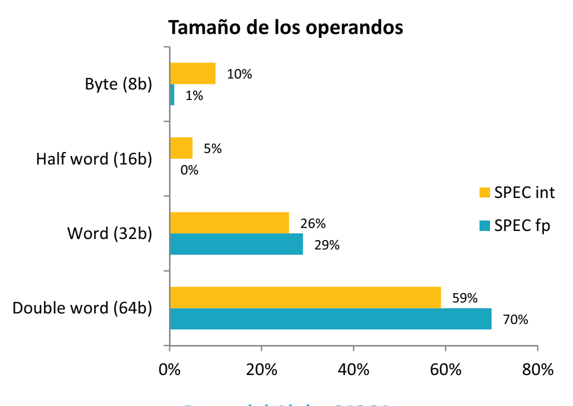
- **Modos de direccionamiento (H&P 1ª ed).**



- Con pocos bits (8) se codifican un 95% de los valores inmediatos. Muchos de ellos son 0 o 1 (45%)
- Con un desplazamiento de 16 bits se cubre el 85% de los casos.
- Con el modo base + desplazamiento (el modo registro indirecto se puede implementar con desplazamiento cero) se cubre un porcentaje muy elevado de casos (85%, 70% y 97%, respectivamente).
- Datos del VAX 11/70

Estudio Cuantitativo Lenguaje Máquina

- **Tipo y tamaño de los operandos (H&P 4ª ed).**
- Tipos de datos soportados a nivel L.M:
  - **Enteros**: complento a 2, diferentes tamaños
  - **Caracteres**: usualmente en ASCII (8 bits), aunque cada vez es más habitual es uso de 16bit Unicode (Internacionalización).
  - **Coma flotante**: IEEE 754 (32 y 64 bits)
  - **Decimal**: packed decimal (4 bits por dígito) o unpacked decimal (8 bits por dígito). Imprescindible para contabilidad. Números que se codifican de manera exacta en decimal no son exactos en binario. Calculadora bc (práctica 5).
- Muchos de los operandos son direcciones



- ¿Qué tamaños han de ser optimizados para acceder de forma más eficiente?
- ¿Tiene sentido que el procesador acceda directamente a memoria para obtener 64 bits? O ¿es mejor hacerlo en 2 accesos?
- ¿Tiene sentido disponer del acceso a byte? Es necesario una red de interconexión compleja.

Estudio Cuantitativo Lenguaje Máquina

- **Tipos de instrucciones (H&P 4ª ed).**
  - Aritméticas y lógicas
  - Transferencia de datos
  - Control de secuencia
  - Sistema
  - Coma Flotante
  - Decimal
  - String
  - Gráficas y multimedia

80x86 (SPECint92)			MIPS (SPECint2000)			MIPS (SPECfp2000)		
1	LOAD	22%	1	LOAD	26%	1	ADD	23%
2	Cond branch	20%	2	ADD	19%	2	LOAD	15%
3	CMP	16%	3	Cond branch	12%	3	LOAD FP	15%
4	STORE	12%	4	STORE	10%	4	STORE FP	7%
5	ADD	8%	5	OR	9%	5	MUL FP	8%
6	AND	6%	6	CMP	5%	6	ADD FP	7%
7	SUB	5%	7	AND	4%	7	LOAD Inm	5%
8	MOV R,R	4%	8	SUB	3%	8	Cond branch	4%
9	CALL	1%	9	XOR	3%	9	SUB FP	3%
10	RET	1%	10	LOAD Inm	2%	10	STORE	2%
Total		96%	Total		93%	Total		89%

Las instrucciones más utilizadas son las más sencillas.

Instrucciones de control de flujo

- **Instrucciones de control de flujo**
  - Call / return: 13%
  - Salto incondicional: 14%
  - Salto condicional: 73%

Para especificar la @destino del salto, lo más adecuado son los saltos relativos al PC → código independiente de la posición.

Posibles instrucciones de salto condicional

- En función de **bits de condición** (x86):
  - ↓ Esta solución restringe el orden de ejecución de las instrucciones.
- **Registros de condición**. Se realiza test sobre un registro cualquiera que tiene el resultado de una comparación previa (Itanium tiene 128 registros de 1 bit):
  - ↑ Es muy simple
  - ↑ No restringe el orden de ejecución
  - ↑ Usa un registro adicional
- Instrucciones que incluyan la **comparación y el salto**
  - ↑ Una sola instrucción
  - ↓ Puede ser mucho trabajo para una instrucción

Nota importante: el 98,9% de los saltos condicionales están precedidos de una comparación o un test.

Observaciones

- Las instrucciones **sencillas** son las **mas ejecutadas**.
- Las instrucciones **complejas** son **difíciles de usar** por el compilador.
- Las instrucciones **complejas** se pueden **sustituir por** secuencias de **instrucciones sencillas**.
- Los modos de direccionamiento **más utilizados** son los **más sencillos**.
- Los modos de direccionamiento **complejos** pueden ser **emulados** por secuencias de instrucciones que además pueden ser **optimizadas**.
- Con **constantes y desplazamientos** de **pocos bits** se cubren la **mayoría** de los casos.

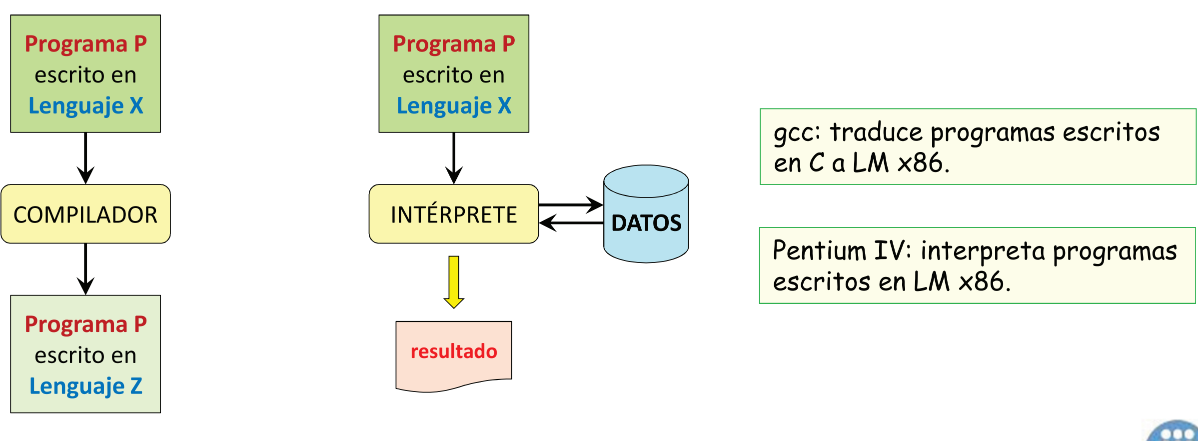
- El objetivo final es hacer que los programas se ejecuten lo más rápido posible.

$$T = N \cdot CPI \cdot Tc$$

RISC vs CISC

- **Niveles de un computador:**
  - Lenguajes de alto nivel (LAN)
  - Lenguaje máquina (LM)
  - Implementación hardware (HARD)

LAN ⇒ compilación ⇒ LM ⇒ interpretación ⇒ HARD

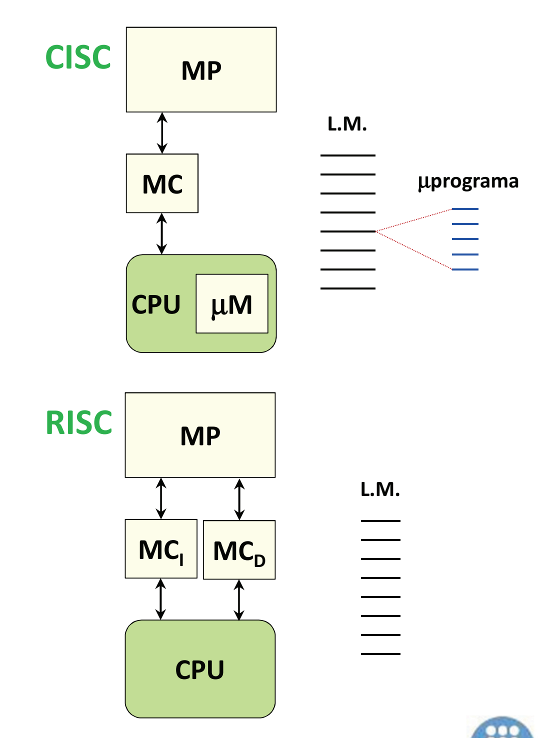


RISC vs CISC

¿Cómo salvar el desnivel entre los Lenguajes de Alto Nivel y el Hardware?

- **CISC: Complex Instruction Set Computer**
  - Instrucciones LM de alto contenido semántico
  - Esfuerzo en interpretación (µcódigo)
- **RISC: Reduced (complexity) Instruction Set Computer**
  - Instrucciones LM de bajo contenido semántico
  - Esfuerzo en compilación

Objetivo: $T = N \cdot CPI \cdot Tc$



RISC versus CISC

CISC	RISC
Instrucciones complejas	Instrucciones sencillas
Instrucciones de tamaño variable	Instrucciones de tamaño fijo
Muchos formatos dependiendo de los operandos	Pocos formatos de instrucciones
Operandos en registro o memoria `ADDL %eax,(%ebx) ;M[ebx]←M[ebx]+eax`	Operandos siempre en registros `ADD Ri,Rj,Rk ;Rk←Ri+Rj`
Cualquier operando de cualquier instrucción puede estar en Memoria.	Instrucciones especiales para acceder a memoria: Load / Store
Históricamente pocos registros	Banco de registros grande (≥32)
Modos de direccionamiento complejos	Modos de direccionamiento simples (Rb+despl.)



RISC versus CISC

CISC (Complex Instruction Set Computers):

CISC MP MC CPU μM L.M. μprograma

- Usaban **instrucciones complejas** para acercar el LM a los LAN.
- La memoria era un recurso **escaso y caro**. La longitud de los programas era uno de los parámetros para evaluar una arquitectura.
- Después de introducir la μprogramación era efectivo **mover funciones** realizadas por una serie de instrucciones **de L.M. hacia el μprograma**: aumentaba la densidad de código, reducía el tráfico con memoria y el tamaño del programa ⇒ incrementaba la eficiencia de la máquina.
- Se suponía que **mover instrucciones** complejas **hacia el LM simplificaría** la tarea del **compilador** al eliminar el desnivel semántico entre el LM y los Lenguajes de Alto Nivel (p.e.: INDEX, CASE, CALLS en VAX 11/70).
- Disponían de **muchos y complejos modos de direccionamiento**.
- Necesitaban **instrucciones de longitud variable** ⇒ dificulta mucho el fetch y la decodificación.
- La μmemoria alcanzó tamaños desorbitados (480 KB en el VAX 11/70) y se convierte en el cuello de botella del sistema. Además no está libre de errores.

RISC vs CISC

RISC (Reduced Instruction Set Computers):

- Los estudios prácticos descubrieron que los procesadores CISC estaban sobredimensionados. Los compiladores no utilizaban eficientemente los modos de direccionamiento e instrucciones de los procesadores CISC.
- Los procesadores RISC utilizan instrucciones muy simples, que al ser generadas por el compilador pueden ser optimizadas.
- Los compiladores generan código más fácilmente y más eficiente para este tipo de máquinas.
- A partir de los años 80, los procesadores empezaron a ser más rápidos que las memorias.

RISC

MP

MC_I MC_D

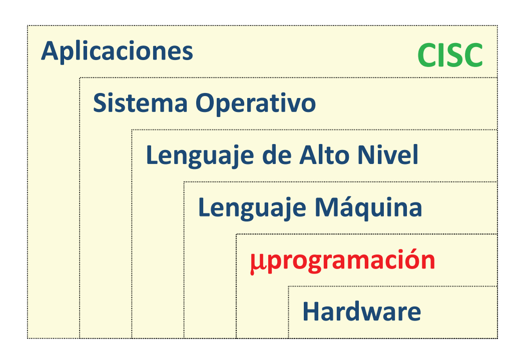
CPU

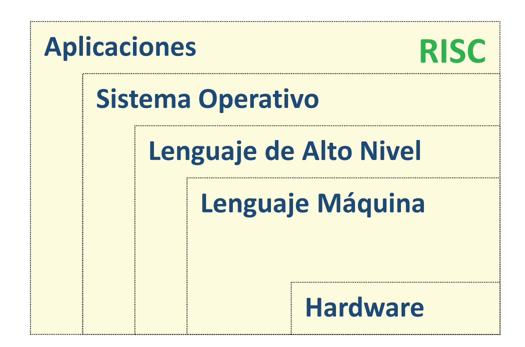
L.M.

- Los procesadores RISC disponen de **muchos registros** de propósito general ⇒ la mayoría de los operandos pueden estar en registros, incluyendo los parámetros y variables locales de las subrutinas.
- Sólo se accede a memoria con load y stores.
- Estas máquinas están pensadas para su ejecución **segmentada** ⇒ se busca **ejecutar 1 instrucción por ciclo**

μprogramación

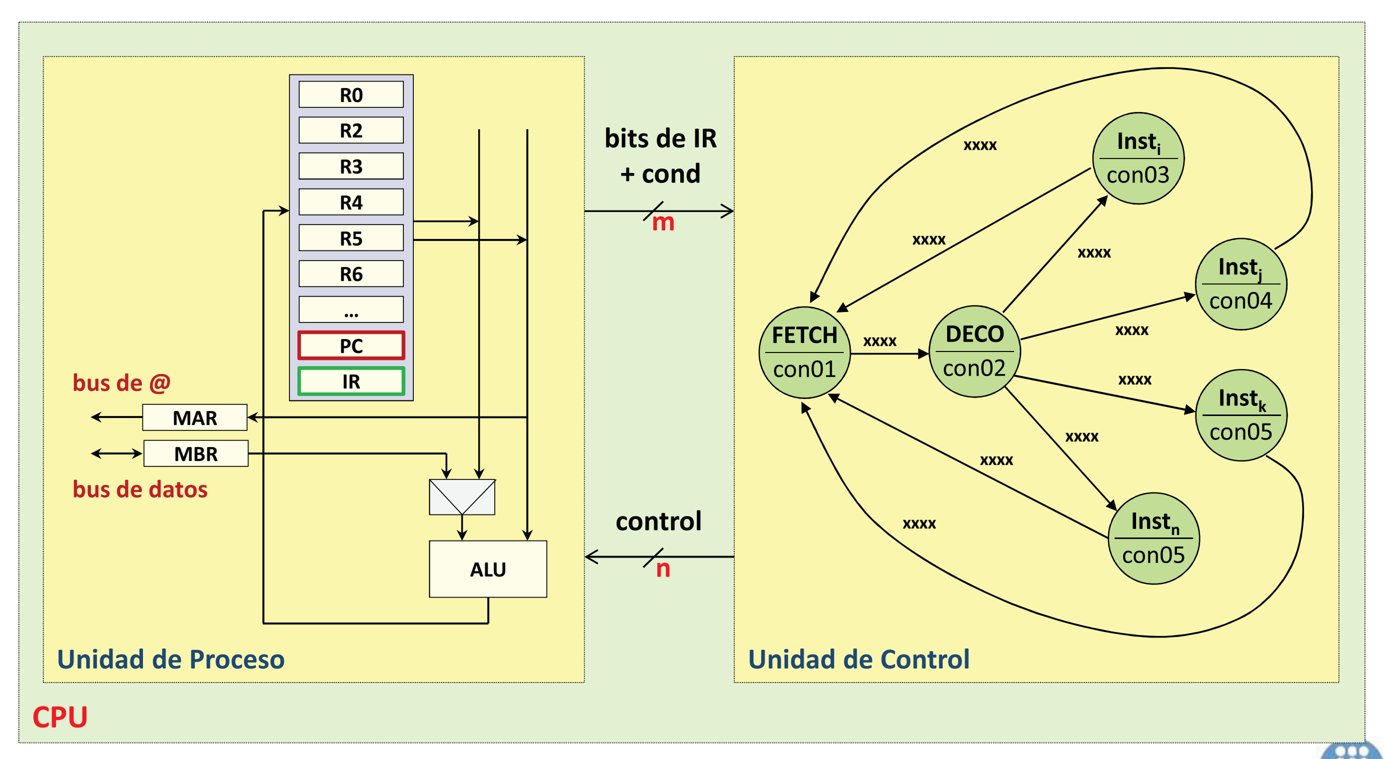
- Técnica utilizada para simplificar el diseño de la unidad de control de los procesadores CISC.
- Visión vertical en niveles de un computador



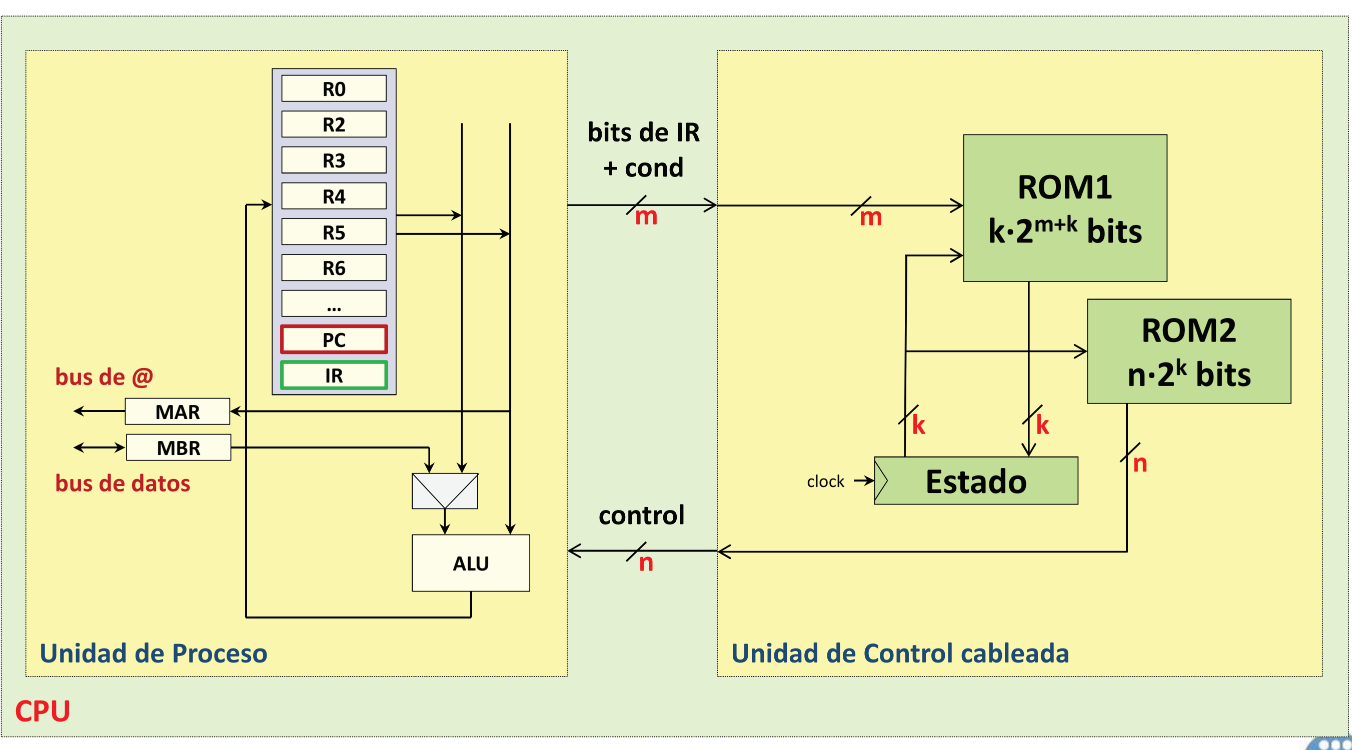


- Elementos básicos de un procesador
  - Unidad de proceso: ALUs, registros, elementos combinacionales, …
  - Unidad de control: sistema secuencial
- Unidad de Control **cableada** versus **μprogramada**

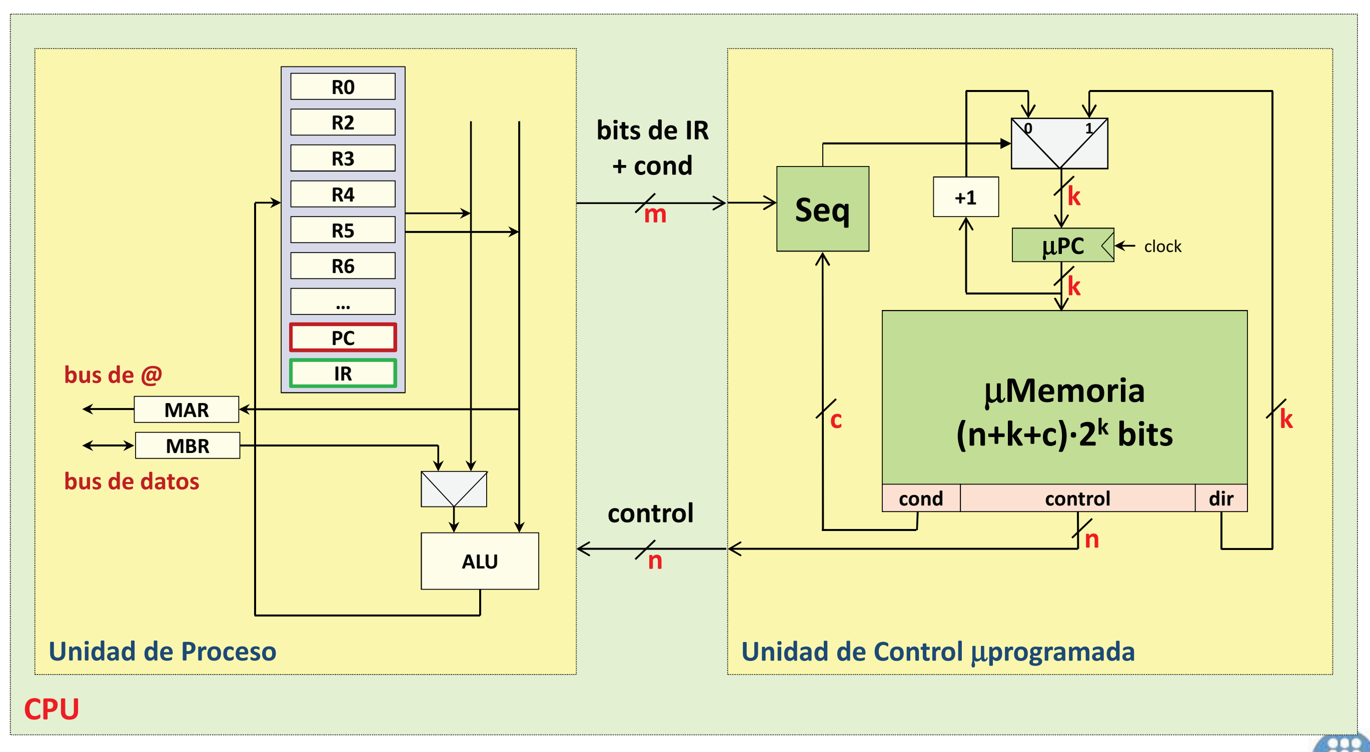
CPU: elementos básicos



CPU cableada



CPU μprogramada



μprogramación

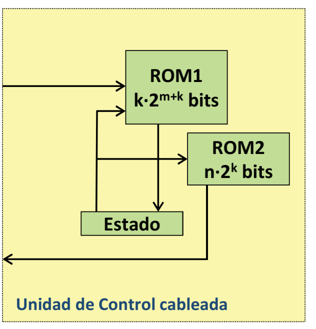
- El contenido de la μmemoria es el μprograma.
- El μprograma está compuesto por μinstrucciones, que equivalen a los estados del sistema secuencial que define la unidad de control.
- En las μ instrucciones se codifican 3 campos:
  - **control**: bits que gobiernan directamente los circuitos de la UP.
  - **dir**: dirección de la siguiente μinstrucción en caso de que sea necesario romper el secuenciamiento implícito del μprograma.
  - **cond**: forma de evaluar los flags y bits de IR para decidir si el salto a **dir** es efectivo.
- En el μprograma se identifican las fases en que se puede dividir la ejecución de una instrucción de Lenguaje Máquina:
  - Fetch
  - Decodificación
  - Ejecución detallada
- **El μprograma es el intérprete del Lenguaje Máquina.**

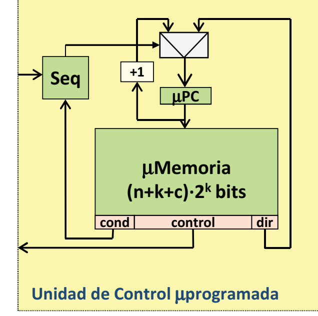
μprograma de la MS

0	MAR = PC; PC = PC+1	#FETCH
1	RD	
2	IR = MBR	
3	IF IR15==1 GOTO 15	#DECODE
4	IF IR14==1 GOTO 10	
5	MAR = SP; SP = SP+1	#ADD
6	RD; MAR = SP	
7	MBR = MBR + A	
8	WR; GOTO 0	
9	MAR = SP; SP = SP+1	#SUB
10	RD; MAR = SP	
11	MBR = MBR – A	
12	WR; GOTO 0	
13	IF IR14==1 GOTO 20	
14	A = SIGN(IR)	#PUSH
15	SP = SP-1	
16	MAR = SP; MBR = A	
17	WR; GOTO 0	
18	MAR = SP; SP = SP+1	#BEQ
19	RD	
20	ALU = MBR; IF Z==0 GOTO 0	
21	A = SIGN(IR)	
22	PC = PC+A; GOTO 0	

Unidad de Control Cableada vs μprogramada

- **Unidad de control cableada**
  - X estados, k =$\log_2$ X
  - Entradas: m bits (IR + cond)
  - Salidas: n bits (control de la UP)
  - ROM1: $k \cdot 2^{m+k}$ bits para calcular el siguiente estado
  - ROM2: $n \cdot 2^k$ bits para obtener las señales de control asociadas a cada estado.
- **Unidad de control μprogramada**
  - X μinstrucciones (estados), k =$\log_2$ X
  - Entradas: m bits (IR + cond)
  - Salidas: n bits (control de la UP)
  - Cond: c bits para seleccionar la siguiente μinstrucción.
  - μmemoria: $(n+k+c) \cdot 2^k$ bits para obtener las señales de control asociadas a cada estado.
  - Seq: 2^{c+m} bits para seleccionar la siguiente μinstrucción.





- **Ejemplo:**
  - 1024 estados, k = 10,
  - Entradas: 14 bits (m = 14)
  - Salidas: 24 bits (n = 24)
  - Cond: 8 condiciones (c = 3)
- **Coste UC cableada**
  - ROM1: $10 \cdot 2^{24}$ = 167.772.160 bits
  - ROM2: $24 \cdot 2^{10}$ = 24.576 bits
- **Coste UC μprogramada**
  - μmemoria: $37 \cdot 2^{10}$ = 37.888 bits
  - Seq: 2^{17} = 131.072 bits

Procesadores x86

- Las instrucciones x86 son complejas y difíciles de implementar.
- Las primeros procesadores que ejecutaban x86 eran μprogramados.
- En las implementaciones actuales, se utiliza un **traductor hardware** que traduce de x86 a μoperaciones.
- Las μoperaciones tienen una complejidad similar a instrucciones RISC:
  - R1 ← R4 + EDX
  - EAX ← R5 - 8
  - R3 ← M[EAX+EBX*2-48]
  - M[R3] ← R9
- La CPU ejecuta las μoperaciones como si fuera un procesador RISC

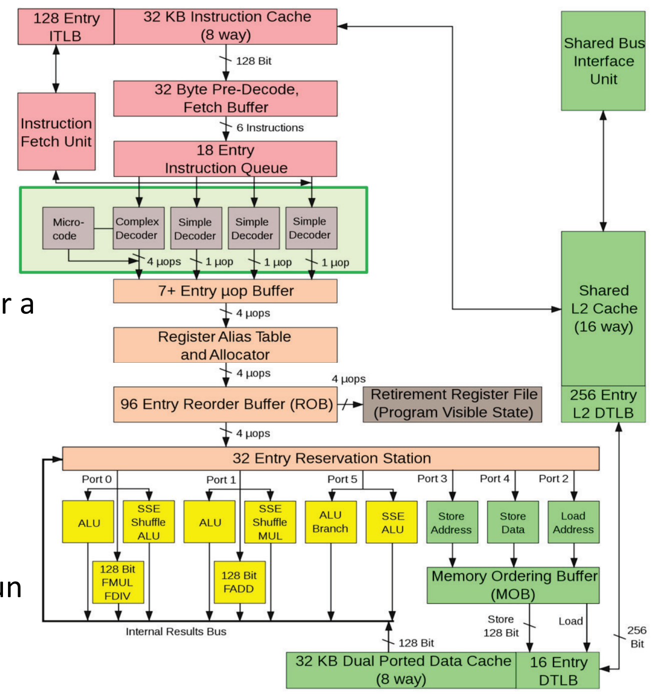


Imagen obtenida en wikipedia.org

Ejemplos de traducción de x86 a μops

ADDL $17, 36(%EBX, %EDX, 4)
R1 ← M[EBX+EDX*4+36]
R2 ← R1+17
M[EBX+EDX*4+36] ← R2

RET
EIP ← M[ESP]
ESP ← ESP+4

CALL $1
ESP ← ESP-4
M[ESP] ← EIP
EIP ← EIP+despl. ($1)

PUSHL 100(,%EBX,8)
R1 ← M[EBX*8+100]
ESP ← ESP-4
M[ESP] ← R1

COMPILACIÓN vs INTERPRETACIÓN

- ¿Cómo se ejecuta un programa Java?

